A Semantic Web Services Approach Towards Automated Software Engineering

Ulrich Dinger
TU Dresden
Hans-Grundig-Str. 25
01062 Dresden, Germany
dinger@rn.inf.tu-dresden.de

Roy Oberhauser
Aalen University
Beethovenstr. 1
73430 Aalen, Germany
roy.oberhauser@htw-aalen.de

Christian Reichel
ETH Zurich
Universitystr. 6, CAB F57
CH-8092 Zurich
christian.reichel@inf.ethz.ch

Abstract

The growing complexity of software, combined with demands for greater productivity and shorter cycles, creates an increasing demand for more automation and integration within the software engineering (SE) domain. When viewed holistically, the heterogeneous nature, implicit feature cross-dependencies, and manual administration of the toolchain infrastructure results in unnecessary complexity, inefficiencies, and reduced reliability for the SE process. A common infrastructure is missing that provides an interoperable and distributed tool environment, addresses feature dependency selection, and automates toolchain workflow composition and execution. To address these challenges, this paper explores the practicality of a unifying Semantic Web Services approach towards Automated Software Engineering (SWS-ASE).

1. Introduction

The W3C, OASIS and several organizations, such as the SWSI, SWSS, DAML group and the WSMO organization, have developed various specifications and approaches in order to support Service Oriented Architectures (SOAs)/Web Services (WS) and the Semantic Web vision [1]. The basic principles of the Semantic Web vision are applicable and beneficial to certain challenges facing the software engineering (SE) domain, especially with regard to expressing a subset of SE information in a machine-processable (and thus automatable) form [2].

For the SE domain, there is a continuous and growing demand to produce software in shorter cycles with higher productivity and quality. Currently, heterogeneous toolchains (e.g, MDA and AOP frameworks, build and deployment tools) common in SE creates difficulties when viewed holistically from an Automated Software Engineering (ASE) perspective. For instance, the large set of explicit and implicit feature choices made during various SE process stages mean that their effects are not always apparent. This is often due to implicit dependencies elsewhere in the toolchain workflow that in turn affect both SE product and process qualities. The lack of machine-processable feature classifications spans technology and design choices to ISO 9126 quality attribute tradeoffs. A further issue is that the manual steps (or manually-created toolchain workflows) between tools become brittle, include implicit assumptions, and are time-consuming to adapt. Semantic information about what a tool does, the transformations it performs, what input formats are expected, what output formats are generated, etc. are not easily accessible or machine-processable. A third issue is the lack of standardization, distribution, and efficient asset management in the tool arena (e.g., search, compatibility issues, integration, and update notifications), making tool software assets costly, and the process error-prone.

In order to improve this situation, a comprehensive and unifying approach called SWS-ASE is explored that automates the selection and composition of tool functionality. The key ideas are to attach semantic information to existing SE tools, to create a tool registry of distributed and local services, to use reasoning engines for automatic composition and, therewith, to bootstrap Semantic Web Services for the SE process of WS-based applications. Using such an approach yields significant benefits by:

1) Amalgamating heterogeneous, distributed tool functionality in a standardized way
2) Improving feature dependency selection support using a classification system
3) Automating toolchain workflow composition and execution

While it is unrealistic to expect complete automation under all circumstances, SWS-ASE enhances the engineer's capabilities and provides CASE-like benefits to the SE domain, especially with regard to usability, reliability, and efficiency.

2. The SWS-ASE domain model

Ontologies for the SE domain are an ongoing research area [3]. The SWS-ASE ontology is currently divided into four sub-ontologies. The first one represents the sub-

domain of target technologies that relate to execution environments. The second ontology introduces the "feature" concept which is able to describe certain characteristics or qualities of a piece of software such as those defined in the ISO 9126 standard. The "specification wrapper" ontology is the third ontology and provides the semantic links (wrappers) for existing programming languages and standards (such as Java, BPEL, and the Service Language Layer (SLL) [4] described later). The fourth ontology describes the set concepts that relate to the "application logic" concept, e.g., workflow parts such as asynchronous or synchronous calls. In order to decorate an application logic part (e.g., within a Java document) with specific characteristics, a feature list (e.g., security, profiling, etc.) can be attached.

3. SWS-ASE process

Essentially, the complete SWS-ASE process comprises five steps: service registration, knowledge retrieval, request assembling, workflow planning, and workflow execution. First, the service provider generates and publishes a semantic service description in the *service registry*. A reliability check validates this description. At a later point in time, a *requestor* (such as an engineer, deployer) formulates a request (with the help of a software tool) that results in a semantic request that is submitted to the semantic *reasoner*. The *reasoner* then retrieves the information of all known services (using the *service registry*) and plans the workflow(s) that fulfills the semantic request. The user can then submit the planned workflow for execution by a workflow engine and receives the result.

3.1 Service registration

The implementation of the solution approach is based on a SOA with WS. Thus, all created WS encapsulate a certain piece of functionality that can be reused within the SE process. Additionally, each service is annotated with semantic information, which describes the pre-/post-conditions (PRC/POC) and effects (E) of a specific service operation. This information is stored in the *service registry*.

As an example for registered SWS, the AddSecurityService of Listing 1 provides semantic information for its addSecurity operation using SLL [4], which was developed as a special-purpose, platform independent WS programming language, e.g., to provide better mapping capabilities to different target platforms and technologies.

As shown in line 30, the operation precondition uses a rule from the SE domain that states that the input concept I1 (SLL program) must not have the *message encryption* feature. Additionally, the effect section states that the output O1) inherits all features from the input (line 36) and the feature *message encryption* is added (line 38).

```
01 // register the XML-based SLL model
02 namespace xsll = "http://fxl-project.com/lang/sll/1.1/xsll.xsd";
03 // register the domain ontologies
04 namespace dm = "http://fxl-project.com/ontologies/dm.owl";
05 // register domain specific rules
06 namespace rdm = "http://fxl-project.com/rdm.rml";
07 //…
08 service AddSecurityService {
09
10 // this operation adds security capabilities to the input sll service
11 public operation addSecurity($i1 as xsll:unit, $i2 as sec:configuration)
12    returns ($o1 as xsll:unit) {
13
14    // semantic description
15    semantic {
16      signature ($I1 as dm#sll, $I2 as dm:sec_config)
17          returns ($O1 as dm#sll);
18
19       mappings {
20        //mapping from semantic to input paramters
21        $i1 = $I1/attachment;  //Note: OWL properties with complex type
22        insert $I2/type into $i2/encryption;
23        //…
24        //mapping from output parameters to semantic
25        insert $o1 into $O1/attachment;
26       }
27
28       pre_conditions {
29         // prohibit the '#message_encryption' feature for the input I1
30         rdm:prohibitFeatures($I1, ['dm#message_encryption']).
31         //…
32       }
33
34       effects {
35         // the output O1 inherits all features of the input I1
36         rdm:copyFeatures($I1, $O1).
37         // add '#message_encryption' feature
38         rdm:addFeatures($O1, ['dm#message_encryption']).
39         //…
40       }
41
42       post_conditions {//n.a.}
43     }
44     // application logic
45     // …
46   }
47 }
```

Listing 1. AddSecurityService

Table 1 shows some entries of the service registry. Along with the service name, operation-specific semantic input and output ids and concepts (cp. the semantic signature of listing 1) are provided. The semantic description includes the type as well as the facts/rules. The following abbreviations of domain specific rules are used in the last column: *supportFeatures (sF), prohibitFeatures (pF), hasMinimumOneFeature (hMOF), copyFeature(cF*) and *addFeature(aF)*. All other services of the service registry are described similarly.

3.2 Request assembling

In order to automatically convert and deploy a theoretical Service (BookService), the requestor formulates the desired objective as a semantic request. The tuple $Rq=(G, S, T, F)$ represents a request, where $G=(C, E)$ is the directed graph, $S \subseteq C$ the set of start concepts, $T \subseteq C$ the set of target concepts with $T \cap S = \emptyset$, and F the set of desired features. In the case of a deployment request for the BookService, the following elements could be assigned:

$S=\{dm\#sll, dm\#config\}$

$T=\{dm\#report\}$

$F=\{dm\#message\_encryption, dm\#profiling,$
$dm\#async\_call, dm\#deployment/target=\#dm:JBoss\}$

Table 1. Service registry entries

service name	semantic inputs		Semantic outputs		semantic	
	id	concept	id	concept	type	rules/facts
BPEL2SLL	I1	bpel	O1	sll	PRC	sF(I1, [‘workflow’,...])
	I2	config			E	cF (I1, O1)
SLL2EJB	I3	sll	O2	depl_config	E	cF (I3, O3)
	I4	config	O3	xjava*	E	aF (O3, [‘#ebj2.0’])
SLL2WSDLS	I5	sll	O4	wsdls	E	cF (I5, O4)
JavaCompile	I6	xjava*	O5	xclass*	E	cF (I6, O5)
AddTracing	I7	xjava*	O6	xjava*	PRC	pF (I7, [‘#op_call_tracing’])
					E	cF (I7, O6)
					E	aF (I7, [‘#op_call_tracing’])
AddSecurity	I8	sll	O7	sll	PRC	pF (I8, [‘#message_encryption’])
	I9	config			E	cF (I8, O7)
					E	aF (O7, [‘message_encryption’])
Bundling	I12	xclass*	O9	bundle	PRC	hF (I12, [‘#ejb2.0’])
	I13	wsdls			PRC	pF (I12, [‘#bundled’])
					PRC	cF (I12, O9)
					PRC	aF (O9, [‘#bundled’])

3.3 Workflow planning

In the context of Web service composition, ongoing research into the fundamental problems of workflow planning include AI planning [5] and the automatic composition of WS [6]. In the domain of SE, two typical scenarios occur that must be covered by an algorithm: a) The calculation of all workflow alternatives for a specific request b) The calculation of an optimal workflow for the request that fulfills a certain criterion (best calculation time, shortest workflow, etc.).

However, both planning problems are NP-hard. To provide better performance, the underlying planning algorithm can use case-specific strategies, including Hill-climbing and heuristic methods. For example, in the current SWS-ASE implementation, the evaluation of the planning state to approximate the distance between the initial states *S* and the target states *T* is realized via an integrated Relaxed Graphplan heuristic [7]. In addition to heuristics, additional information is calculated for each state (e.g., the set of helpful rules *R*). This reduces the branching factor for typical SE scenarios where a fast initial result is needed. Furthermore, specific restrictions (e.g., max. 10 composite services, timeouts) and prohibitions for certain workflow patterns (permutations, sequential execution of independent services) are made.

3.4 Workflow execution

The result of the preceding planning phase is the generated graph W=(C, E) for a specific semantic request, which connects a set of WS from the service registry. Therefore, the graph W can be represented as a WS written, e.g., in BPEL or SLL [4]. One generated workflow for the aforementioned *BookService* deployment request is illustrated in Figure 2.

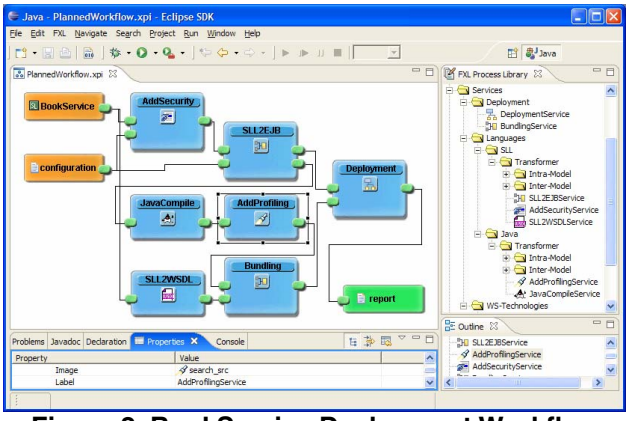

Figure 2. BookService Deployment Workflow

Assuming the BookService was modeled in UML and converted to SLL, the SLL document together with a configuration document serve as input *S* to the workflow (upper left boxes) where the output *T* is shown as a report box. In the first step, the *AddSecurityService* is called in order to weave in the *message_encryption* feature (as an aspect) in the SLL program. The *SLL2WSDLService* generates a WSDL-S description from the SLL service input. In parallel, the SLL program is transformed into an EJB that comprises a set of Java files. This set of Java files is compiled via the *JavaCompileService*. The ProfilingService instruments the class files to enable the requested profiling capabilities. All class files along with the WSDL-S are bundled via the *BundlingService*. In the last step, the EJB bundle is deployed and registered using the *DeploymentService*. The output of the last service is a concept called *report* containing information about the deployment action.

3.5 Feature configurator

Figure 3 shows the graphical user interface for the specification of desired application features (feature modeling) in the context of workflow planning and execution. The given input (BookService, configuration) and desired output concepts (report) of a specific SE planning request can be added as boxes on the left. In this state, the composite SE process is not yet planned and available (cp. Figure 2). On the right, the feature configurator is shown, which allows a tree-based and explicit selection of the desired application features. When the feature selection is completed, the “Generate

workflow(s)" link launches the SWS-ASE process and automatically generates a valid workflow based on the given strategy parameters.

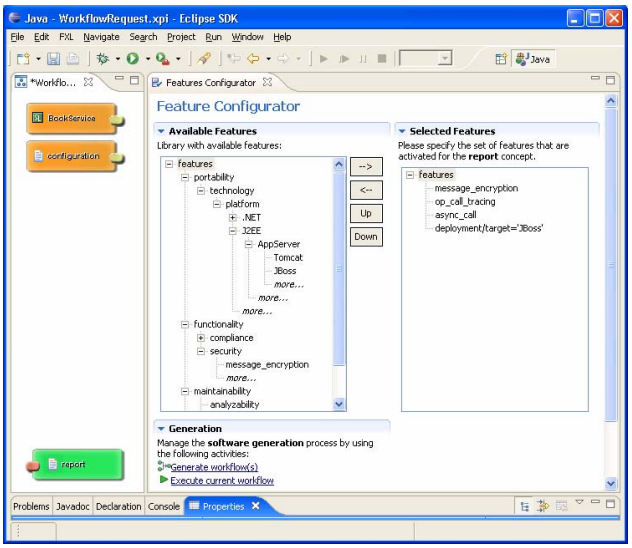

Figure 3. Feature configurator view

4. Results and conclusion

The SWS-ASE results show that such an approach yields promising quality improvements for common SE tasks (such as automated toolchain composition and execution) and that major parts of the SWS-ASE underlying vision can already be applied. By preparing and registering tool functionality as SWS, an automated SE process was realized that is able to provide a way of amalgamating heterogeneous, distributed tool functionality in a standardized way. In this context, user-specific views such as the *Feature Configurator* were created to improve usability by hiding technology-specific details and enabling feature-driven application development.

SWS-ASE provides the capability for indicating the direction and degree of influence to quality characteristics of any given feature choice (given the appropriate ontology and rules), thus illuminating the impact of a set of feature choices to the overall quality. For example, the choice of asynchronicity potentially enhances scalability and efficiency, while potentially reducing response time and maintainability. These quality tradeoffs can thus be weighed more explicitly by an engineer, and various quality priorities and optimizations are selectable to achieve a certain quality goal. Thus, compared with a manual SE process, fewer deployment and runtime errors occur due to the inherent dependency and relation checks within the creation phase (reasoning) of the SE toolchain.

To measure efficiency, various measurement sets were executed, including response time and scalability measurements for two typical search problems that occur in the planning phase:

a) Find the first workflow that fulfills the request.
b) Find the set of all possible workflows.

In this context, the scalability measurement sets, which are not illustrated here, showed that the calculation of *all* possible workflow combinations would cause unacceptable results for a typical registry size of 50 services. However, an initial/next workflow variant was available after an acceptable period of time (e.g., 0.92sec for 50 services) sufficient for most SE planning tasks. Additionally, SWS-ASE shows that typical SE feature selections can be reasonably handled when the appropriate algorithms and heuristics are applied together with constraints that reduce the set of possible combinations. Fundamentally, the enhanced automation of the SE process yields improved usability, reliability, and efficiency.

Some inherent risks of the SWS-ASE are that the overall process quality strongly depends on the correctness of the semantic descriptions, and that the number of relationships (regarding possible I/O connections, feature cross-dependencies, etc.) increases exponentially with the number of involved services. Nevertheless, the SE domain has much to gain from the SWS vision and the enhanced automation capabilities it could bring. By improving the integration of the software tool infrastructure with semantically-enhanced tools while planning and automating workflows based on desired features, the SWS-ASE approach shows that significant benefits are realizable that support and enhance the quality of software engineering products and processes.

5. References

[1] Berners-Lee, T., Fischetti, M.: "Weaving the Web – The original design and ultimate destiny of the World Wide Web, by its inventor, Harper San Francisco; 1st edition, September 1999.

[2] Tetlow, P., et al., "Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering", W3C Working Draft, Oct. 2005.

[3] Wongthongtham, P., "Software Engineering Ontologies and Their Implementation", *Proceedings of the IASTED Conference on Software Engineering*, 2005, pp. 208-213.

[4] Kossmann, D., Reichel, C., "SLL: Running my Web Services on Your WS Platforms", *Proceedings of ICWS 2005*, Industry Track, July 2005. Also see www.open-xl.org.

[5] Medjahed, B., et al. "Composing Web services on the Semantic Web," *The VLDB Journal*, Volume 12, Issue 4, November 2003, pp. 333 – 351.

[6] B. Aleman-Meza, I. B. Arpinar, and R. Zhang, "Automatic Composition of Semantic Web Services", in *Proceedings of ICWS03*, pgs: 38-41, Las Vegas, USA, 23-26 June, 2003.

[7] J. Hoffmann, B. Nebel, "The FF Planning System: Fast Plan Generation Through Heuristic Search", *Journal of Artificial Intelligence Research*, Volume 14, 2001, pp. 253 - 302.